Assignment 2

Chi Yin Wong 836872

Question 1

a. Signature function

```
# a function to calculate the exponent of very large numbers – will be used in the functions defined in this question
def power(base,power,mod):
   result = 1
   while power > 0:
      if power % 2 == 1:
         result = (result * base) % mod
      power = power // 2
      base = (base * base) % mod
   return result

# signature function
def sign(M,n,d):
   signature = power(M,d,n)
   return signature
```

b. Verification function

```
# verification function
def verify(S,n,e,M):
   M_1 = power(S,e,n)
   if M_1 == M:
      return True
   return False
```

c. Blinding function

```
def blindsign(x,n,M,e,d):
   M_b = (power(x,e,n)*M)%n
   s_b = power(M_b,d,n)
   s = (s_b/x)%n
   return s
```

d. Both d and e were also submitted in the template for marking

```
542014982849169332005219868369260490979899527151971580501181774601
313377100929104556546510432988398662469813448416376660199714651
775104469310305340866396358529781696259603117497592710346046567774
517257413042685154118831000024416
```

e. 113369226439623876134770215651761851186768559754878264850904455285747573985687952080622736368282688894283482025559280217141929695863109886024401924726393873986358780905578460554412863546441327980317647022070101231296979685867520

Question 2

Approach A does not successfully authenticate that the message M originally came from Alice. If an attacker intercepts and modifies the ciphertext C, since S and C are concatenated, S will also be modified. Bob will receive the modified ciphertext C’, and then decrypt with his private key to obtain M’ and S’. Bob will then use Alice’s public key and M’ to check if he gets S’, which he will, so he will not think that an attacker sent the message.

Approach B on the other hand can successfully authenticate that the message M originally came from Alice. Since Alice is encrypting the message with Bob’s public key, Bob can retrieve the message using his private key. However, Alice uses her public key to sign the message, so the only way for Bob to verify that the message was sent from Alice is to compute the signature himself with the public key of Alice and the ciphertext C and see if it matches the S that was sent to him. However, an attacker can modify ciphertext C and send Bob C’. When Bob uses Alice’s public key with C’, he will obtain a different value for S. Thus, he will think that the attacker sent the message instead of Alice, and the attacker has tricked Bob into thinking they sent it. We can show this as follows:

Alice has M, sends both C and S to Bob where

C = Encrypt(K_b, M)

S = Sign(K_a, C)

An attacker intercepts and modifies C → C’ (eg. By changing some bits) and sends C’ to Bob

Bob receives C’ and S

Bob can only decrypt C’ but he cannot decrypt S because he does not have Alice’s private key. Thus, Bob must calculate S with Alice’s public key and the ciphertext he received.

Bob calculates Sign(K_a, C’) which will not be equal to S

The attacker has tricked Bob into thinking it was sent from them (the attacker) and not Alice

Question 3

a. Variable input size

   This hash function satisfies the requirement. Given that the message M can be broken down into blocks of predefined fixed sizes M1, M2, … , Mm, the message M can be a variable length. The larger M is, the more blocks there will be (i.e. m increases).

b. Fixed output size

This hash function satisfies the requirement. Message M is broken into blocks of predefined sizes M1, M2, … Mm. Each block is then encrypted and then converted to binary to be XORed with the next block. Since each block is in mod n, we can assume that their binary conversion is of the same length (eg. If n = 257, 1 can be represented by 00000001 and 256 can be represented by 11111111 which both have length 8). Thus, it must be the case that the output size is of a fixed length.

c. Efficiency

This hash function satisfies the requirement. A has function is considered efficient if it can be computed within polynomial time complexity. The primary operation in this hash function is the bitwise XOR operator. Assuming we can only compute 1 bit at a time and that we have n bits (i.e. length of the message is n), the hash function will have complexity O(n) which is less than polynomial time. Thus, the hash function is efficient.

d. Preimage resistant

The hash function satisfies this property. A hash function is preimage resistant if, given a hash value h, it is computationally infeasible to find m such that H(m) = h. In this hash function, there are multiple instances of XOR's made up of blocks of the original message m, which mean that given a final hash value consisting of 1's and 0's, it is infeasible to find the bits that make up the message blocks M1,M2, …, Mm as there are so many combinations that can result in the final hash value. Thus, the hash function is preimage resistant.

e. Second preimage resistant

The hash function does not satisfy this property. A hash function is second preimage resistant if given a message m and corresponding hash value H(m), it is computationally infeasible to workout another input m' in the message space that results in H(m') = H(m). However, a simple example can show that this is false. Consider m = 1111 0000 and m' = 0000 1111 (after encryption). The message m is separated into 2 blocks, m1 = 1111 and m2 = 0000, and these two blocks will be XORed. The hash value for m is H(m) = 1111. Similarly, the message m' is separated into 2 blocks, m1' = 0000 and m2' = 1111, and the resulting hash value after XOR'ing the blocks will be H(m') = 1111. Since H(m) = H(m'), the hash function does not satisfy the second preimage resistant property.

f. Collision resistant

The hash function does not satisfy this property. Consider a simple example where, after encryption, we have

M1 = 11111111 00000000 11111111

M2 = 00000000 11111111 11111111

We split these into 8 bit blocks to input into the hash function i.e.

For Message 1:
H(11111111, 00000000, 11111111)
= 11111111 XOR 00000000 XOR 11111111
= 00000000

For Message 2:
H(00000000, 11111111, 11111111)
= 00000000 XOR 11111111 XOR 11111111
= 00000000

Since H(M1) = H(M2), it is computationally feasible to find a pair M1,M2 such that this is true. Thus, the hash function doesn’t satisfy the collision resistant property.